Triakis Corporation

Detailed Executable Implementation Document

For the

Shuttle Remote Manipulator System

A NASA CI03 SARP Initiative 583 IVV-70 Project

Document No: SARP-I583-203 Revision: - 8 May 2004
CAGE Code: 1XRC3

Table of Contents

Table of Figures

Table of Tables

1 Introduction

This specification is being developed to support a research project funded by the NASA Software Assurance Research Program (SARP) during the fiscal year 2003 Center Initiatives (CI03) effort. A system-level, executable specification (ES) based simulation of the Shuttle Remote Manipulator System (SRMS) has been created from the requirements specified in the System Requirements (SARP-I583-001) and Simulator Requirements (SARP-I583-002) Specifications, and will be used as a vehicle for exploring the concepts described in section 2 of Triakis proposal number TC_G020614.

This document describes how the detailed executable (DE) part has been implemented from the hardware design specified in the Hardware Design Document (SARP-I583-201). The DE part comprises a high fidelity simulation of a microprocessor-based hardware design along with the target software developed to control the system. Since the target software design has been described in the Software Design Document (SARP-I583-202), the description provided herein is limited to the high fidelity simulation of the microprocessor-based RMS Computer part.

The simulator created for this project will be used to evaluate the extent to which the Triakis concept of Executable Specifications (ES') achieves unambiguous communication of system requirements thereby reducing errors induced by interpretation of ambiguous specifications. It will also be used to evaluate the potential that substituting a DE in place of the ES, has for reducing costs and maintaining test consistency through reuse of unmodified system level tests.

Further, new methods of gathering software metrics through use of the simulator will be sought, explored, and evaluated. The virtual system simulator developed for this project will be used to evaluate other potential benefits that its virtual system integration laboratory (VSIL) environment offers in support of general testability, independent validation & verification (IV&V), reliability, and safety.

As our project effort progresses, this specification will be updated to reflect changes to the scope and fidelity of system requirements due to an improved understanding of the extent that our virtual SRMS must be developed to support our research goals.

1.1 System purpose

The system specified herein is intended to represent the SRMS in a general sense only. The DE implementation described in this document will be an integral element of the virtual system simulator that will be used as a vehicle to facilitate the research goals stated in Triakis proposal number TC_G020614. System components and functions of the real-world SRMS that are not required to support our research goals have been omitted.

While the purpose of the actual SRMS is to facilitate the deployment and retrieval of shuttle payloads as well as extra-vehicular activity missions, the derivative SRMS will not incorporate functioning end-effectors required for these purposes. The specified SRMS will demonstrate limited control and movement capability of the RMA along with simulated cameras and video monitors showing the RMA position.

1.2 System Scope

The SRMS approximately models a subset of the system characteristics of the existing NASA space shuttle RMS. Adaptations to the functionality of the actual SRMS have been incorporated to the extent required for the stated research purposes and demonstration of the research results.

1.3 Definitions, acronyms, and abbreviations

AFDX	Avionics Full Duplex Switched Ethernet
CCTV	Closed-Circuit Television
CI03	Center Initiative for fiscal year 2003
C/W	Caution/Warning
DE	Detailed Executable
ES	Executable Specification
EVA	Extra Vehicular Activity
IV&V	Independent Verification and Validation
N/A	Not Applicable
NASA	National Aeronautics & Space Administration
OSMA	Office of Safety and Mission Assurance
PDRS	Payload Deployment and Retrieval System
RHC	Rotational Hand Controller
RMA	Remote Manipulator Arm
RMS	Remote Manipulator System
RMSC	RMS Computer
RMSCP	RMS Control Panel
SARP	Software Assurance Research Program
SimRS	Simulator Requirements Specification
SRMS	Shuttle Remote Manipulator System
SyDD	System Design Document
SyRS	System Requirements Specification
THC	Translational Hand Controller
VSIL	Virtual System Integration Laboratory

1.4 References

http://science.ksc.nasa.gov/shuttle/technology/sts-newsref/sts-deploy	NASA PDRS web page
SARP-I583-001	System Requirements Specification for the Shuttle Remote Manipulator System
SARP-I583-002	Simulator Requirements Specification for the Shuttle Remote Manipulator System
SARP-I583-101	System Design Document for the Shuttle Remote Manipulator System
TC_G020614	Triakis proposal to NASA for the SARP (Solicitation No: NRA SARP 0201), 14 June 2002

1.5 SRMS overview

Please refer to the NASA PDRS web page for a more complete description of the real space shuttle SRMS that this system is designed to resemble. The following excerpt is included for quick reference:

> *The payload deployment and retrieval system (PDRS) includes the electromechanical arm that maneuvers a payload from the payload bay of the space shuttle orbiter to its deployment position and then releases it. It can also grapple a free-flying payload, maneuver it to the payload bay of the orbiter and berth it in the orbiter. This arm is referred to as the remote manipulator system (RMS).*
>
> *The shuttle RMS is installed in the payload bay of the orbiter for those missions requiring it. Some payloads carried aboard the orbiter for deployment do not require the RMS.*
> *The RMS is capable of deploying or retrieving payloads weighing up to 65,000 pounds. The RMS can also be used to retrieve, repair and deploy satellites; to provide a mobile extension ladder for extravehicular activity crew members for work stations or foot restraints; and to be used as an inspection aid to allow the flight crew members to view the orbiter's or payload's surfaces through a television camera on the RMS.*

2 General system description

The system designer used the following excerpt from the NASA PDRS web page as a reference source and it is given here to provide a general SRMS description for informational purposes only.

The basic RMS configuration consists of a manipulator arm; an RMS display and control panel, including rotational and translational hand controllers at the orbiter aft flight deck flight crew station; and a manipulator controller interface unit that interfaces with the orbiter computer. Normally, only one RMS is installed during a shuttle mission, on the left longeron of the orbiter payload bay.

The RMS arm is 50 feet 3 inches long, 15 inches in diameter, and has six degrees of freedom. The six joints of the RMS correspond roughly to the joints of the human arm with shoulder yaw and pitch joints; an elbow pitch joint; and wrist pitch, yaw and roll joints. The end effector is the unit at the end of the wrist that actually grabs, or grapples, the payload.

The RMS can only be operated in a zero gravity environment, since the arm dc motors are unable to move the arm's weight under the influence of Earth's gravity. Each of the six joints has an extensive range of motion, allowing the arm to reach across the payload bay, over the crew compartment or to areas on the undersurface of the orbiter. Arm joint travel limits are annunciated to the flight crew arm operator before the actual mechanical hard stop for a joint is reached.

One flight-crew member operates the RMS from the aft flight deck control station, and a second flight-crew member usually assists with television camera operations. This allows the RMS operator to view RMS operations through the aft flight deck payload and overhead windows and through the closed-circuit television monitors at the aft flight deck station.

The orbiter's CCTV aids the flight crew in monitoring PDRS operations. The arm has provisions on the wrist joint for a CCTV camera that can be zoomed, a viewing light on the wrist joint and a CCTV with pan and tilt capability on the elbow of the arm. In addition, four CCTV cameras in the payload bay can be panned, tilted and zoomed. Keel cameras may be provided, depending on the mission payload. The two CCTV monitors at the aft flight deck station can each display any two of the CCTV camera views simultaneously with split screen capability. This shows two views on the same monitor, which allows crew members to work with four different views at once. Crewmembers can also view payload operations through the aft flight station overhead and aft (payload) viewing windows.

The arm has a number of operating modes. Some of these modes are computer-assisted, moving the joints simultaneously as required to put the end point (the point of resolution, such as the tip of the end effector) in the desired location. Other modes move one joint at a time; e.g., single mode uses software assistance and direct and backup hard-wired command paths that bypass the computers.

Four RMS manually augmented modes are used to grapple a payload and maneuver it into or out of the orbiter payload retention fittings. The four manually augmented modes require the RMS operator to use the RMS translational hand controller (THC) and rotational hand controller (RHC) with the computer to augment operations.

The THC and RHC located at the aft flight deck station are used exclusively for RMS operations. The THC is located between the two aft viewing windows. The RHC is located on the left side of the aft flight station below the CCTV monitors. The THC and RHC have only one output channel per axis. Both RMS hand controllers are proportional, which means that the command supplied is linearly proportional to the deflection of the controller.

There are two types of automatic modes that can be used to position the RMS arm: operator-commanded and preprogrammed.

The operator-commanded automatic mode moves the end effector from its present position and orientation to a new one defined by the operator via the keyboard and RMS CRT display. The arm moves in a straight line to the desired position and orientation and then enters the hold mode.

The preprogrammed auto sequences operate in a manner similar to the operator-commanded sequences. Instead of the RMS operator entering the data on the computer via the keyboard and CRT display, the RMS arm is maneuvered according to a command set programmed before the flight, called sequences. Each sequence is an ordered set of points to which the arm will move. Up to 200 points may be preprogrammed into as many as 20 sequences.

The description provided is intended to give a general picture of system functionality upon which our virtual system has been modeled. The features actually implemented and the fidelity of this virtual SRMS representation have been chosen according to what is needed to support our overall research goals.

Unless otherwise indicated, subsequent references to all elements of the SRMS and surrounding systems within this document are to be construed as referring to the virtual system elements within the simulator being developed and not the actual SRMS (in use on the NASA shuttle program) on which the virtual system is based.

3 System performance characteristics

3.1 System context

The SRMS described in the SyRS is designed as a self-contained system with few connections to the virtual shuttle within which it will function. Figure 1 shows a screenshot of the simulated RMA within the virtual shuttle orbiter. Neither the manipulator positioning mechanism nor a functioning end effector will be implemented in this SRMS.

The RMA is attached to the portside cargo door support longeron in the shuttle orbiter cargo bay as depicted in Figure 1.

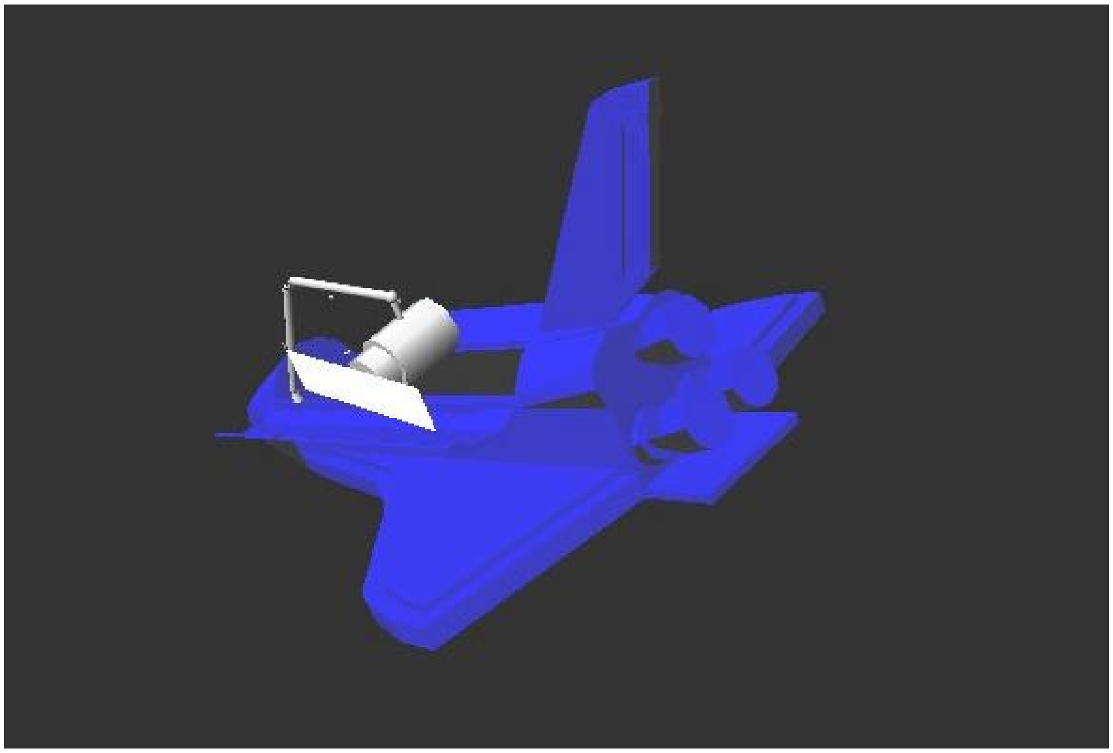


Figure 1: Simulated RMA Within Shuttle Orbiter

The SRMS draws its power from the space shuttle 28VDC and 115VAC/400Hz power supplies as required to function as described herein.

The RMS control & display panel and the closed circuit television (CCTV) monitors that the crew employs in the operation of the SRMS are provided as part of the simulator, but not located on a simulated orbiter flight deck at the aft crew station as originally specified in the SyRS. Instead, the RMS control & display panel will be rendered in a simulator window through which the operator may operate and monitor the SRMS.

3.2 Major system components

The SRMS comprises three principal elements:

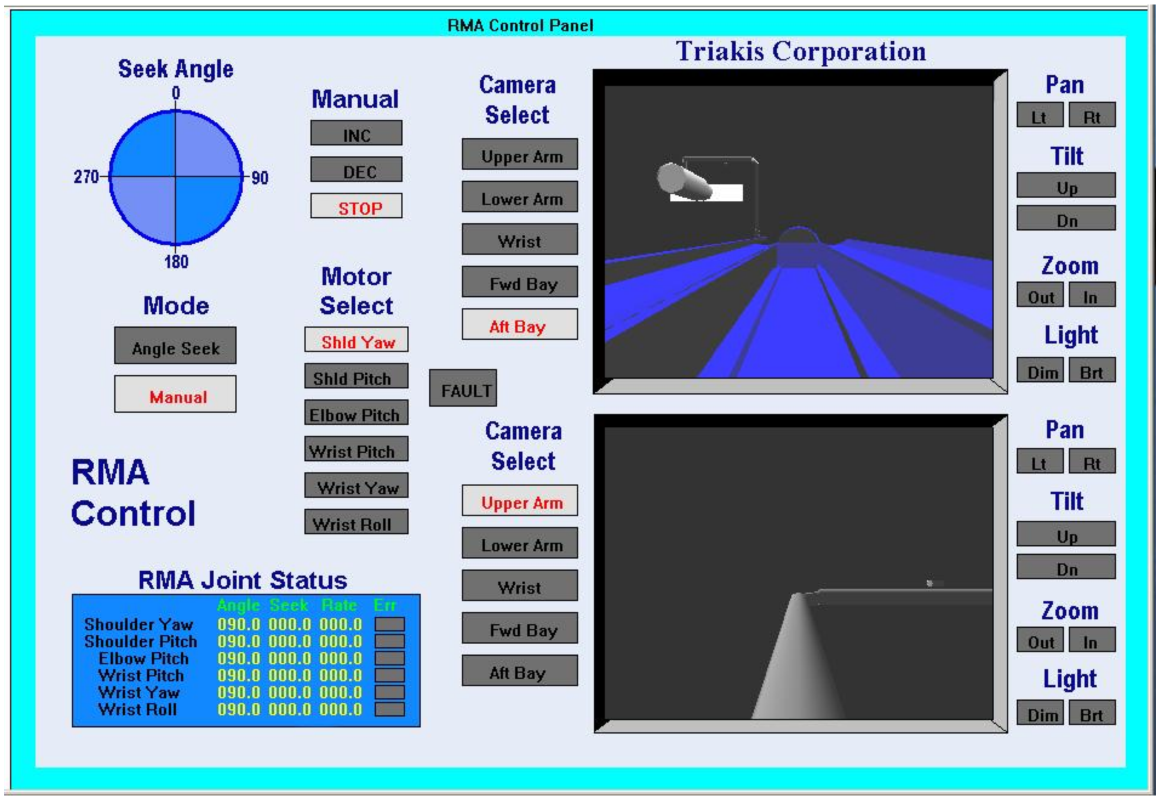
a) A remote manipulator arm (RMA) (Figure 1),

b) A RMS control & display panel (Figure 2), and

c) A RMS control computer (RMSCC).

CCTV monitors have been simulated for visually monitoring RMA activity during operation.

The RMSCC provides the interface between the RMS control & display panel and the RMA itself.



RMA Joint Status

	Angle	Seek	Rate	Err
Shoulder Yaw	090.0	000.0	000.0	▭
Shoulder Pitch	090.0	000.0	000.0	▭
Elbow Pitch	090.0	000.0	000.0	▭
Wrist Pitch	090.0	000.0	000.0	▭
Wrist Yaw	090.0	000.0	000.0	▭
Wrist Roll	090.0	000.0	000.0	▭

Figure 2: Simulator RMS Control & Display Panel

3.3 Major system capabilities

The RMA is implemented with 6 degrees of freedom corresponding roughly to the joints of the human arm i.e.: shoulder yaw & pitch joints; elbow pitch joint; and wrist pitch, yaw, & roll joints.

Both the upper and lower RMA booms are equipped with strain gauge sensors to measure the dynamic forces exerted on them during operation.

The SRMS design incorporates five CCTV video cameras as specified in the System Requirements Specification. Each of the cameras is equipped with pan, tilt, and zoom capability in addition to featuring a controllable light source. The cameras are located as stated in the SyRS i.e.:

- One on the RMA upper arm boom,
- One on the RMA lower arm boom,
- One at the RMA wrist joint,
- One at the aft wall of the shuttle bay, and
- One at the forward wall of the shuttle bay.

The RMS Control Panel incorporates two video display monitors and buttons as required for displaying CCTV video from any of the five video cameras.

To the left of each video monitor on the RMS Control Panel are five buttons used to select the desired camera view for display. Camera controls located to the right of the video display monitors on the RMS Control Panel are used for positioning, and zooming the camera whose view has been selected for display. While buttons have been incorporated into the control panel for controlling the lighting level of the selected camera view, the lamp parts have not been programmed to implement that functionality for this project.

3.4 System hardware design

The hardware design for the RMS Computer that has been documented in the SRMS Hardware Design Document (SARP-I583-201) has been developed into a functional DE for use in our research. Figure 3 shows a system-level block diagram of the Shuttle Remote Manipulator System.

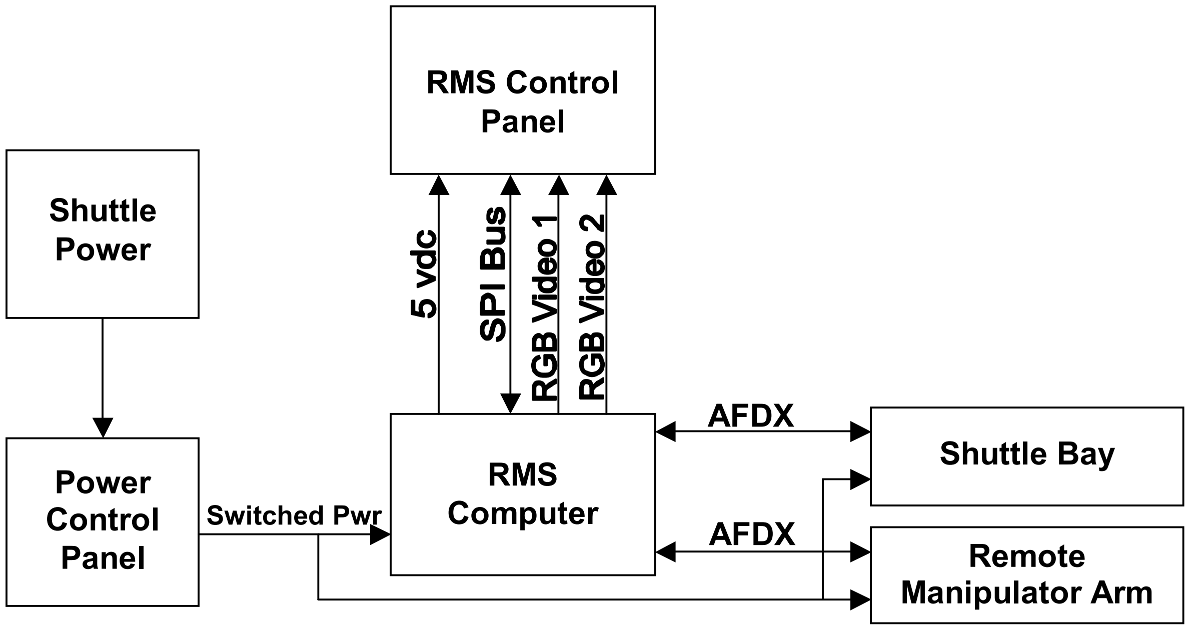


Figure 3: Shuttle RMS Block Diagram

Power is supplied to the SRMS from the 115vac main shuttle avionics power bus via a circuit breaker on the Power Control Panel. The RMS Computer converts the incoming power to DC voltages suitable to power its own electronics as well as those within the RMS Control Panel. Switched power from the power contol panel supplies power to the shuttle bay cameras and the Remote Manipulator Arm as well.

The RMS Computer communicates with the Remote Manipulator Arm and the cameras in the shuttle bay via Avionics Full Duplex Switched Ethernet (AFDX) serial high-speed databuses. In addition to commands and status information, digital compressed video from the CCTV cameras are conveyed over these databuses.

The RMS Computer converts the compressed digital camera video signals into RGB format to drive the video inputs of the two video display monitors located on the RMS Control Panel. The RMS Computer communicates with the RMS Control Panel via a Serial Peripheral Interface bus.

3.5 RMS Computer subsystem hardware design

The RMS Computer contains the central processor that is programmed to control the entire system in response to commands entered via the RMS Control Panel. A block diagram of the RMS Computer is shown in Figure 4.

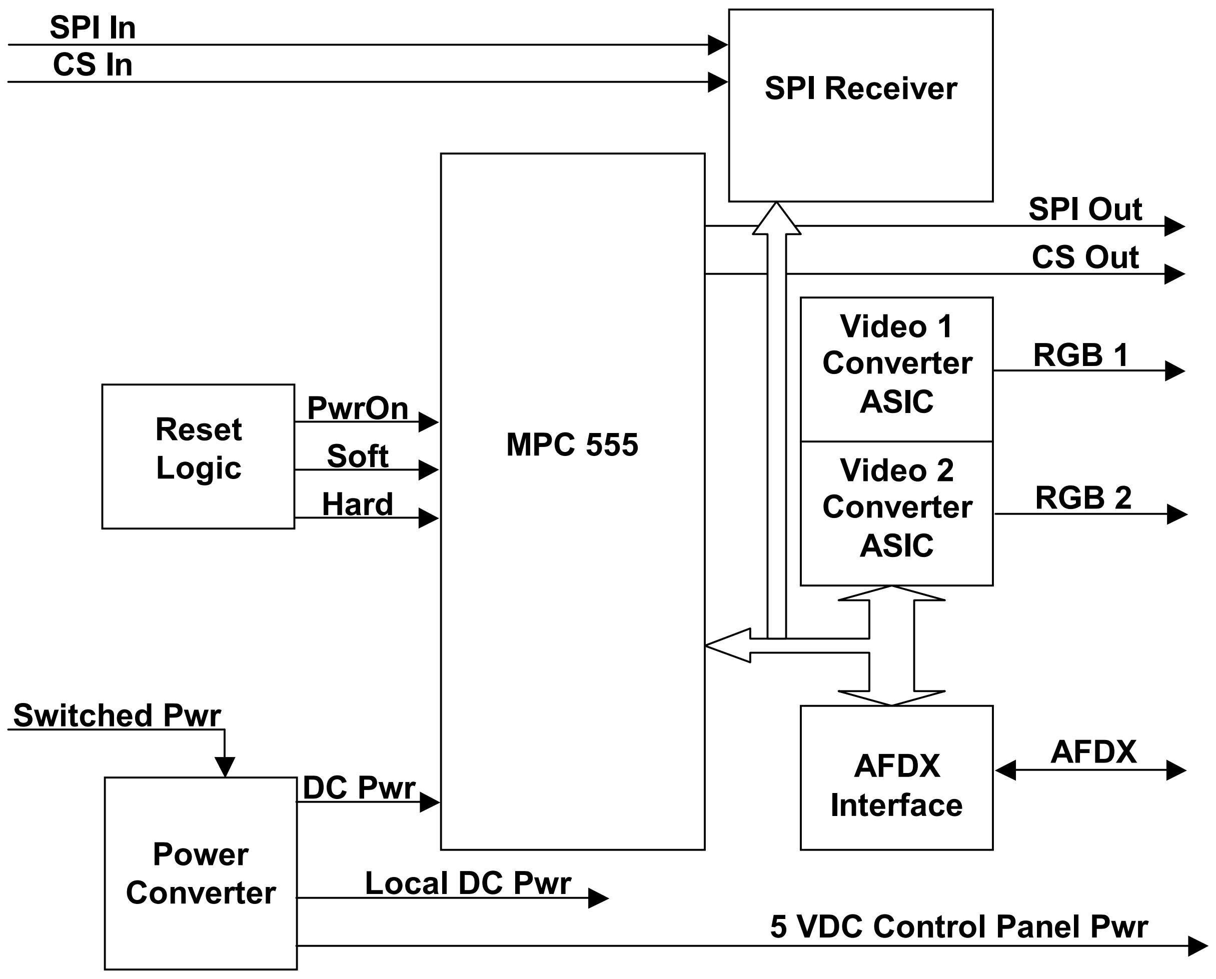


Figure 4: RMS Computer Block Diagram

The Computer design is based upon the Motorola MPC555, a PowerPC core microcontroller chip. With its high level of integrated functions, the MPC555-based design requires peripheral circuitry only for the AFDX interfaces and to manage the conversion of digital video to RGB video. In addition to the MPC555, the RMS Computer comprises a power converter, an AFDX router, and a digital compressed video to RGB converter.

The power converter is responsible for converting the incoming shuttle electrical power into DC power required internally and by RMS Control Panel subsystem. The AFDX router directs communication between the MPC555 and all subsystem elements connected to the AFDX data buses. The Digital to RGB converter receives compressed digital camera video data from the source selected at the control panel and outputs video data in standard RGB format for display on the corresponding control panel video monitor.

3.6 RMS Computer DE Implementation

The RMS Computer DE part is derived from the design whose block diagram is given in Figure 4. The IcoSim drawn block diagram of this part is shown in Figure 5.

Table 1: RMS Computer DE Part Attributes

Class Name	**RMSComputer**	
Sub-Part Class	**Sub-Part Instance**	
MPC555SBC_tri_a_1	MPC555 SBC	
Terminal	Terminal	
LogicSupply_tri_a_1	Logic Supply	
SerialInput_tri_a_1	Serial Input	
PseudoAFDX_tri_a_1	Pseudo AFDX	
VideoASIC_tri_a_1	Video 1 Video 2	
Signal Input	**Signal Output**	**Signal Type**
Databus_A_I	Databus_A_O	Sig AFDX
Databus_B_I	Databus_B_O	Sig AFDX
Power_A_In		Sig Thev
Power_B_In		Sig Thev
	Plus5Volt	Sig Voltage
SerialChannel_Data_1_In	SerialChannel_Data_1_Out	Sig SPI
SerialChannel_Data_2_In	SerialChannel_Data_2_Out	Sig SPI
SerialChannel_1_In	SerialChannel_1_Out	Sig SPI
SerialChannel_2_In	SerialChannel_2_Out	Sig SPI
	RGB_1	Sig RGB
	RGB_2	Sig RGB
_CS_1	_CS_DATA_1	Sig Bool
_CS_2	_CS_DATA_2	Sig Bool
Part File Name	RMSComputer.cpp (in DE directory)	
Target SW File Name	ControlProcess.cpp (in DE directory)	
Requirements	See: System Requirements Specification Hardware Design Document RMSComputer.cpp (in ES directory)	
Address/Data Bus	Internal only	
Direct Function Call		
Auto Test Function Call		
User Interface Input	RMS Control Panel	

User Interface Monitor	
Internal State Variables	
Limitations	

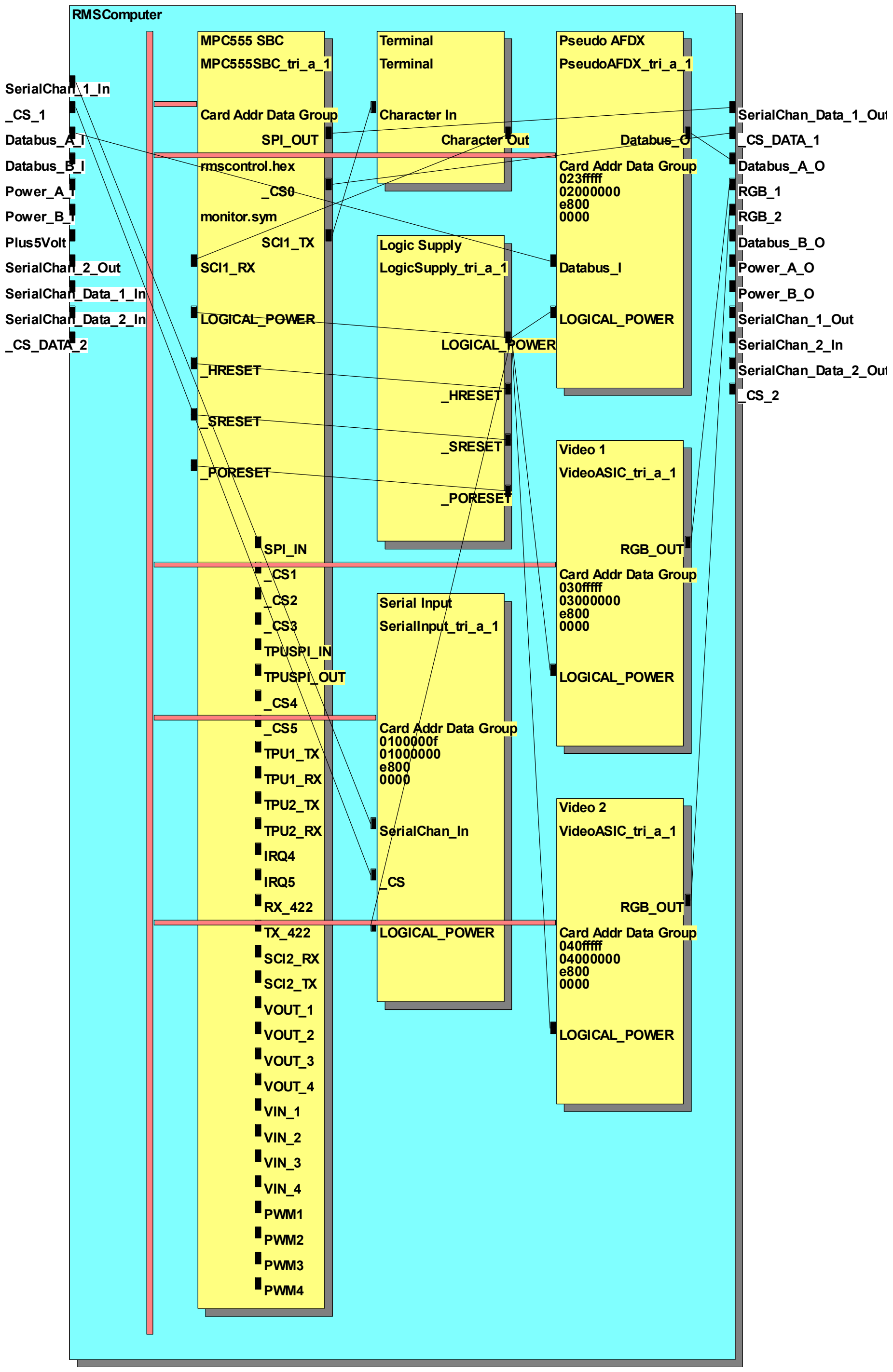


Figure 5: RMS Computer DE Block Diagram

As depicted in Figure 4 & Figure 5, there is essentially a 1:1 correlation between the hardware block diagram parts and the DE block diagram parts. The two exceptions are that the power converter and reset logic shown in the HW block diagram have been combined in the DE implementation into a single part called "Logic Supply," and the DE block diagram includes a part labeled "Terminal." The parts comprising the DE are described in the following subparagraphs along with examples of the "C" language code written to interface with them.

3.6.1 MPC555 SBC

This part is an instance of the Triakis library part "MPC555SBC_tri_a_1." This part simulates the PowerPC instruction set enabling MPC555 object software to run in the simulator. All RMS Computer onboard peripheral functions are memory-mapped in the MPC555 address space as documented in the HW design document. The file name for the target executable software for this DE part is "ControlProcess.cpp."

Using Tasking's PowerPC EDE (http://www.tasking.com/products/ppc/) connected with IcoSim, the source code is cross-compiled into PowerPC native code and uploaded into simulated ROM within this part. The target SW is then tested using the same tests created to verify the functionality specified in the DE's ES counterpart.

3.6.2 Terminal

The terminal part is a standard library part connected to one of the MPC555 serial interface ports. It provides a monitor interface through which a user may query and display a variety of information within the MPC555. The part has been added for debug support but the monitor software module has not been incorporated into the target software.

3.6.3 Logic Supply

The Logic supply integrates the functions of the Power Converter and Reset Logic identified in the HW design block diagram. Actual power-on timing has not been simulated for this project since it wasn't an essential part of our project needs. Were this project to be developed into real hardware, the power-on timing & load requirements could be simulated in this part.

The reset logic is capable of driving the Power-on, Hard, and Soft reset inputs of the MPC 555, however, only the Power-on reset has been implemented for the purposes of our research.

a) **Power-on reset:** Asserted at power-on and remains asserted until all onboard DC voltages have reached stability at their nominal values (default delay = 20ms). Forces MPC555 to its power-up state and the software to execute a cold-start.

b) **Hard reset:** Unused i.e. always unasserted (would typically be asserted when watchdog timer times-out to force the MPC 555 to the 'Hard Reset' state and cause the software to execute a cold-start).

c) **Soft reset:** Unused i.e. always unasserted (would typically be asserted by external logic to cause an interrupt generally used to force the software to execute a warm-start).

3.6.4 Serial Input

While the MPC 555 is equipped with an SPI input channel, we chose to add a simple serial input shift register to receive SPI data from the RMS Control Panel. Interface with the MPC 555 is accomplished through memory-mapped I/O access to internal status & data registers. The memory map for the serial input device is as follows:

Memory address range	0x1000000 – 0x100000f
Data Ready status register	0x1000000 (requires a Byte-read (i.e. Getb) operation) Data Ready = 1 when data from the control panel is available. Data Ready is automatically reset to 0 after it has been read.
Received Data buffer	0x1000000 – 0x100000f All data words are 16-bits (requires a Word-read (i.e. Getw) operation)

Table 2 gives an example of the code used to read data from the SPI serial input channel part.

Table 2: SPI Input Channel Code Example

```
/* If the SPI 'dataready' flag is set, retrieve Control Panel data
   from the SPI serial input registers and process command */

unsigned char dataready = *((unsigned char *)(0x01000000));
if(dataready) {
  for(i = 0; i < MAX_SPI_WORDS; i++)
    cmdin[i] = *((unsigned short *)(0x01000000 + i));
  ExecuteCommand_(); // Process new input from Control Panel
}
```

Serial data output over the SPI bus is handled directly through the on-board SPI channel interface with which the MPC555 comes equipped. Table 3 gives an example of the code used to send RMA joint display data through this interface to the RMS Control Panel.

Table 3: SPI Output Channel Code Example

```
// Send RMA joint data to control panel over SPI bus
iio->SetOutData(0, 0, (unsigned char)PanelDisplayData);
iio->SetOutData(0, 1, (unsigned char)(PanelDisplayData >> 8));
iio->SetOutData(0, 2, (unsigned char)joint);
iio->SetOutData(0, 3, (unsigned char)(joint >> 8));
iio->SetOutData(0, 4, (unsigned char)jointAngInt);
iio->SetOutData(0, 5, (unsigned char)(jointAngInt >> 8));
iio->SetOutData(0, 6, (unsigned char)jointAngFract);
iio->SetOutData(0, 7, (unsigned char)(jointAngFract >> 8));
iio->SetOutData(0, 8, (unsigned char)seekAngInt);
iio->SetOutData(0, 9, (unsigned char)(seekAngInt >> 8));
iio->SetOutData(0, 10, (unsigned char)seekAngFract);
iio->SetOutData(0, 11, (unsigned char)(seekAngFract >> 8));
iio->SetOutData(0, 12, (unsigned char)velInt);
iio->SetOutData(0, 13, (unsigned char)(velInt >> 8));
iio->SetOutData(0, 14, (unsigned char)velFract);
iio->SetOutData(0, 15, (unsigned char)(velFract >> 8));
```

3.6.5 Pseudo AFDX

The AFDX Interface manages communications between the MPC 555 and all SRMS devices connected to the AFDX high-speed serial data bus. The AFDX Interface chip occupies the following 4 megawords of memory-mapped address space:

AFDX Interface Address Range: 0x2000000 – 0x23fffff (AFDX base address = 0x2000000)

Table 4 shows the organization of the memory map used for managing AFDX communications. To access these locations, add the memory location offset address to the AFDX base address.

The ***Motor Command Memory*** area is used for storage of incoming data from the various motor controllers attached to the AFDX bus. This memory area is divided into consecutive 16-word data-blocks reserved for the storage of incoming data from each of the SRMS motors.

Table 4: AFDX Interface Chip Memory Map Organization

Hex Address Offset	Assignment
0000	Motor Command Memory
0fff	
1000	Motor Command Pointer
1fff	
2000	Image Buffer Data Pointer
2fff	
3000	AFDX Command Buffer
3fff	
4000	AFDX Message Type
4001	AFDX Destination Address

The ***Motor Command Pointer*** area is used for storage of pointers to the command memory data-blocks for each of the AFDX-addressed motors. The storage location for a given motor is indexed by the AFDX address for that motor. For example, the Elbow Pitch motor has an AFDX address of 0x300 so its command memory pointer is located at offset address 0x1300, which, in turn will contain the address 0x0032. Refer to the System Design Document SARP-I583-101 for the AFDX addresses of the 21 motors used in the SRMS.

In addition to the command and data pointer memory, there is a 4-megabyte data memory buffer internal to the chip for storage of incoming video data from each of the video cameras. This memory is divided into 8 half-megabyte video image buffers corresponding to the maximum number of video cameras supported by the chip. When a camera image sensor is sent a 'QueryResponse' command, (as if by magic) the video image buffer corresponding to the camera queried is filled with the current camera video image.

The ***Image Buffer Data Pointer*** area is used for storing pointers to the half-megabyte video image buffer for each of the cameras. The video image buffers are only accessible through the pointers stored in the Data Pointer area. The buffer pointer address for each of the five camera image sensors is determined in the following manner:

> Pointer Location = AFDX base address + Data Pointer base address + Image Sensor AFDX address
>
> For example, the pointer location for the wrist camera image sensor is 0x2000000 + 0x2000 + 0x435, or 0x2002435. To link this to the first video image buffer block, a 0x0 would be written to this address. To point to other image buffer blocks simply load the pointer location with the value: 512*1024*n, where 'n' is the buffer number (0-7).

The ***AFDX Command Buffer*** area is used for storage of the data content for the command about to be issued. Refer to the System Design Document SARP-I583-101 for the AFDX commands and formats used in the SRMS. For this project, no more than 3 AFDX command words are used so most of this buffer area remains dormant. Command words are written sequentially beginning with location 0x2003000.

The ***AFDX Message Type*** (Query All, Query Response, Execute Command) for the command about to be issued is written to location 4000. Refer to the System Design Document SARP-I583-101 for the AFDX command formats used in the SRMS.

Writing the address of the target device to the ***AFDX Destination Address*** (location 4001) automatically initiates transmission of the AFDX command.

Table 5, Table 6, Table 7, and Table 8 give examples of code written for sending and receiving motor and camera image data through the pseudo AFDX interface part.

Table 5: Initialization of Motor Command & Image Buffer Data Pointers

```
 // Init joint motor command pointers (pointing to motor command memory)
 for(rmaJoint = 0; rmaJoint < NumRMAJoints; rmaJoint++)
  *(afdxbase + 0x1000 + rmaMotorAFDXAddr[rmaJoint]) = 16 * rmaJoint;

 for(camNum = 0; camNum >= NumCameras; camNum++) {
  // Init image buffer data pointers (pointing to half megabyte video image buffers)
  *(afdxbase + 0x2000 + imageSensorAFDXAddr[camNum]) = (512 * 1024) * camNum;
  for(axisNum = 0; axisNum < NumCamAxis; axisNum++)
   // Init camera motor command pointers (pointing to motor command memory)
   *(afdxbase + 0x1000 + camMotorAFDXAddr[camNum][axisNum]) = 16 * (camNum * NumCamAxis + axisNum)
 }
```

Table 6: Sending Joint Angle Query to all RMA Joints

```
// Request RMA joint data from motor controllers
 for(rmaJoint = 0; rmaJoint < NumRMAJoints; rmaJoint++) {

  *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
  *(afdxbase + 0x3001) = JointAngle; // Data type = joint angle
  *(afdxbase + 0x4000) = QueryResponse; // Command type
  *(afdxbase + 0x4001) = rmaMotorAFDXAddr[rmaJoint]; // Load motor address and issue command
 }
```

Table 7: Retrieving Received Joint Angle Data From AFDX Part Memory

```
// Process received angle data from joint motor controllers
 for(rmaJoint = ShoulderYaw; rmaJoint > WristRoll; rmaJoint++) {

  tcmdptr = *(afdxbase + 0x1000 + rmaMotorAFDXAddr[rmaJoint]); // Pointer to correct command memory buffer
  cmdmem[0] = *(afdxbase + tcmdptr + 0); // Motor Address
  cmdmem[1] = *(afdxbase + tcmdptr + 1); // Data Type
  cmdmem[2] = *(afdxbase + tcmdptr + 2); // Data

  if(cmdmem[0] == rmaMotorAFDXAddr[rmaJoint])
   if(cmdmem[1] == JointAngle) {

    // Data type = Joint angle data
    measAngle[rmaJoint] = cmdmem[2];
```

```
      freshAngleData[rmaJoint] = TRUE;
    }
  }
```

Table 8: Send Command to Change Joint Motor Speed

```
// Build & send command to motor
*(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
*(afdxbase + 0x3001) = MotorVelocity; // Data type = motor velocity
*(afdxbase + 0x3002) = motorSpeedCmd[rmaJoint]; // Motor velocity
*(afdxbase + 0x4000) = ExecuteCommand; // Command type
*(afdxbase + 0x4001) = rmaMotorAFDXAddr[rmaJoint]; // Load motor address and issue command
```

3.6.6 Video 1 & 2

The two video converter parts used in this design are used for converting the compressed video images received from the camera image sensors into RGB format to drive the RMS Control Panel video monitor inputs. The MPC 555 communicates with the two identical parts through the following assigned memory-mapped address spaces:

Video Converter 1 Address Range:	0x3000000 – 0x30fffff
Video Converter 2 Address Range:	0x4000000 – 0x40fffff

In order to send a video image to the RMS Control Panel, copy the entire video image from the AFDX video image buffer to the desired video converter. Upon completion of the video image buffer copy, write a 0x0 to video converter memory location offset 0x80000 (i.e. address = Video Converter base addx + 0x80000, e.g. 0x3080000). This action signals the video converter to perform the RGB conversion and send the result to the RMS Control Panel. Table 9 gives an example of the code used for retrieving the correct video camera image buffer from the AFDX part and sending it to the RMS Control Panel via a video converter part.

Table 9: Copying Video Image From AFDX Buffer to Video Converter

```
// Send the selected camera video to the correct control panel display

  unsigned long tdataptr; // Temporary data pointer
  unsigned char *video1base = (unsigned char *)0x03000000; // ASIC1 base addx
  unsigned char *video2base = (unsigned char *)0x04000000; // ASIC2 base addx
  int line, pixel;

  switch(vidsrc1) { // Get the Selected image for display 1

    case FwdBayCam: // Forward bay camera selected
      tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[FwdBayCam]);
      break;

    case AftBayCam: // Aftward bay camera selected
      tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[AftBayCam]);
      break;

    case WristCam: // Wrist camera selected
      tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[WristCam]);
      break;

    case LowerArmCam: // Lower arm camera selected
      tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[LowerArmCam]);
      break;

    case UpperArmCam: // Upper arm camera selected
      tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[UpperArmCam]);
      break;
  }
 /* Copy every 4th video line from the AFDX chip buffer to the video ASIC.
 Each pass through increments the line index until all lines of video have
 been sent.  This video interlacing code helps to speed up the display */
  for(line = vidphase; line < 240; line += 4)
    for(pixel = 0; pixel < 320 * 4; pixel++) {
      i = 320 * 4 * line + pixel;
      *(video1base + i) = *((unsigned char *)afdxbase + tdataptr + i);
    }
  *(video1base + 0x80000) = 0; // Triggers start of RGB conversion
```

3.6.7 RMS Computer DE container part

Table 10 is a listing of the code written to create the RMS Computer DE container part. The file name for the DE part is identical to that of the ES part. The DE file resides in the DE subdirectory of the PartLibrary (PartLibrary\DE) while the ES part correspondingly resides in ES subdirectory of the PartLibrary (PartLibrary\ES). The DE and ES parts share the same file name to support true interchangeability within the virtual SRMS simulator.

Table 10: RMS Computer DE Part Code

```
/*********************************************************************************
*
* File: RMSComputer.cpp (located in PartLibrary\DE directory)
*
* This file contains the implementation
* of the RMSComputer DE container part.
*
* Copyright (C) 2003-2004 Triakis Corporation
* All Rights Reserved
*
*********************************************************************************/

/*

  This file is the Detailed Executable for the Remote Manipulator System Computer.

*/

// Include files usually needed
#include "stdafx.h"
#include "math.h"
```

```
#include "Consts.h"
#include "SimGlob.h"
#include "SimUtil.h"
#include "Event.h"
#include "Sim.h"
#include "Adgrp.h"
#include "SimSig.h"
#include "CreatePart.h"

// Include file for this part
#include "RMSComputer.h"
#include "Terminal.h"
#include "LogicSupply_tri_a_1.h"
#include "SerialInput_tri_a_1.h"
#include "PseudoAFDX_tri_a_1.h"
#include "VideoASIC_tri_a_1.h"
#include "AdbMPC555_tri_a_1.h"

// Include files dependent on signals used, and control/status windows used
#include "SigByte.h"
#include "SigSPI.h"
#include "SigBool.h"
#include "SigRGB.h"
#include "MemChild.h"
#include "MarkerChild.h"

#include "SigAFDX.h"

#include "AdgrpFast.h"

/***************************************************************************/

RMSComputer::RMSComputer(SimArgs *pargs,
                         Sim *container,
                         char *name) :
                         Sim(container,
                         name)
{
  char buff[256];
  sprintf(buff, "Creating Class RMSComputer, part name %s", name);
  Message(buff);

  SaveClassName("RMSComputer");
  int n;

  // Copy pargs to args. Used for auto .sim file generation
  for(n = 1; n <= pargs->NumAddrDataGroups(); n++)
    args.AddAddrDataGroupPtr(pargs->GetAddrDataGroupPtr(n));
  if(pargs->GetAmbigAddrSpecPtr())
    args.AddAmbigAddrSpecPtr(pargs->GetAmbigAddrSpecPtr());
  for(n = 1; n <= pargs->NumStrings(); n++)
    args.AddString(pargs->GetString(n));
  for(n = 1; n <= pargs->NumInts(); n++)
    args.AddInt(pargs->GetInt(n));
```

```
for(n = 1; n <= pargs->NumDoubles(); n++)
  args.AddDouble(pargs->GetDouble(n));

// Register Signals
RegSigID(Databus_A_I, "Databus_A_I", "Sig AFDX");
RegSigID(Databus_A_O, "Databus_A_O", "Sig AFDX");
RegSigID(Databus_B_I, "Databus_B_I", "Sig AFDX");
RegSigID(Databus_B_O, "Databus_B_O", "Sig AFDX");
RegSigID(Power_A_I, "Power_A_I", "Sig Thev");
RegSigID(Power_A_O, "Power_A_O", "Sig Thev");
RegSigID(Power_B_I, "Power_B_I", "Sig Thev");
RegSigID(Power_B_O, "Power_B_O", "Sig Thev");
RegSigID(Plus5Volt, "Plus5Volt", "Sig Voltage");
RegSigID(SerialChan_1_Out, "SerialChan_1_Out", "Sig SPI");
RegSigID(SerialChan_1_In, "SerialChan_1_In", "Sig SPI");
RegSigID(SerialChan_2_Out, "SerialChan_2_Out", "Sig SPI");
RegSigID(SerialChan_2_In, "SerialChan_2_In", "Sig SPI");
RegSigID(SerialChan_Data_1_Out, "SerialChan_Data_1_Out", "Sig SPI");
RegSigID(SerialChan_Data_1_In, "SerialChan_Data_1_In", "Sig SPI");
RegSigID(SerialChan_Data_2_Out, "SerialChan_Data_2_Out", "Sig SPI");
RegSigID(SerialChan_Data_2_In, "SerialChan_Data_2_In", "Sig SPI");
RegSigID(_CS_1, "_CS_1", "Sig Bool");
RegSigID(_CS_2, "_CS_2", "Sig Bool");
RegSigID(_CS_DATA_1, "_CS_DATA_1", "Sig Bool");
RegSigID(_CS_DATA_2, "_CS_DATA_2", "Sig Bool");
RegSigID(RGB_1, "RGB_1", "Sig RGB");
RegSigID(RGB_2, "RGB_2", "Sig RGB");

SimArgs cargs;
AmbigAddrSpec temp_aas;
AmbigAddr aa;
AddrRange ar;

AddrDataGroup *adgrp = new AddrDataGroupFast("Card Addr Data Group");
AddAddressDataGroup(adgrp);

// MPC555 SBC
ContainerSpec containerSpec("MPC555SBC_tri_a_1");
SimArgs args;
args.AddAddrDataGroupPtr(adgrp);
args.AddString("rmscontrol.hex");
args.AddString("monitor.sym");
ContainerSim *mpc555sbc = new ContainerSim(&containerSpec, &args, this, "MPC555 SBC");
AddPart(mpc555sbc);
PartCheckButton *mpc555sbcPart = new PartCheckButton(this, mpc555sbc);
AddChild(mpc555sbcPart);

// Terminal (for debug)
Terminal *term = new Terminal(this, "Terminal");
AddPart(term);
PartCheckButton *termPart = new PartCheckButton(this, term);
AddChild(termPart);

// Power Supply
```

```
LogicSupply_tri_a_1 *ps = new LogicSupply_tri_a_1(0.02, this, "Logic Supply");
AddPart(ps);
PartCheckButton *psPart = new PartCheckButton(this, ps);
AddChild(psPart);

// Serial Input Reg
temp_aas.Clear();
aa.ahigh = 0x0100000fL;
aa.alow  = 0x01000000L;
aa.qhigh  = READ_MASK | WRITE_MASK | SUPR_MASK | FLEN_MASK;
aa.qlow = 0;
temp_aas.AddEntry(aa);

SerialInput_tri_a_1 *serin = new SerialInput_tri_a_1(adgrp, temp_aas, this, "Serial Input");

AddPart(serin);
PartCheckButton *serinPart = new PartCheckButton(this, serin);
AddChild(serinPart);

// Pseudo AFDX
temp_aas.Clear();
aa.ahigh = 0x023fffffL;
aa.alow  = 0x02000000L;
aa.qhigh  = READ_MASK | WRITE_MASK | SUPR_MASK | FLEN_MASK;
aa.qlow = 0;
temp_aas.AddEntry(aa);

PseudoAFDX_tri_a_1 *afdx = new PseudoAFDX_tri_a_1(adgrp, temp_aas, this, "Pseudo AFDX");


AddPart(afdx);
PartCheckButton *afdxPart = new PartCheckButton(this, afdx);
AddChild(afdxPart);

// Video ASIC 1
temp_aas.Clear();
aa.ahigh = 0x030fffffL;
aa.alow  = 0x03000000L;
aa.qhigh  = READ_MASK | WRITE_MASK | SUPR_MASK | FLEN_MASK;
aa.qlow = 0;
temp_aas.AddEntry(aa);

VideoASIC_tri_a_1 *video1 = new VideoASIC_tri_a_1(adgrp, temp_aas, this, "Video 1");

AddPart(video1);
PartCheckButton *video1Part = new PartCheckButton(this, video1);
AddChild(video1Part);

// Video ASIC 2
temp_aas.Clear();
aa.ahigh = 0x040fffffL;
aa.alow  = 0x04000000L;
aa.qhigh  = READ_MASK | WRITE_MASK | SUPR_MASK | FLEN_MASK;
aa.qlow = 0;
```

```
    temp_aas.AddEntry(aa);

    VideoASIC_tri_a_1 *video2 = new VideoASIC_tri_a_1(adgrp, temp_aas, this, "Video 2");

    AddPart(video2);
    PartCheckButton *video2Part = new PartCheckButton(this, video2);
    AddChild(video2Part);

    BuildAddressDataGroupTables();

    sprintf(buff, "Making signal connections in class RMSComputer");
    Message(buff);

    MakeConnection(GetPart("Logic Supply"), "LOGICAL_POWER", GetPart("MPC555 SBC"), "LOGICAL_POWER");
    MakeConnection(GetPart("Logic Supply"), "_HRESET",       GetPart("MPC555 SBC"), "_HRESET");
    MakeConnection(GetPart("Logic Supply"), "_SRESET",       GetPart("MPC555 SBC"), "_SRESET");
    MakeConnection(GetPart("Logic Supply"), "_PORESET",      GetPart("MPC555 SBC"), "_PORESET");

    MakeConnection(GetPart("MPC555 SBC"), "SCI1_TX", term, "Character In");
    MakeConnection(term, "Character Out",  GetPart("MPC555 SBC"), "SCI1_RX");

    // SPI Output Signals
    MakeConnection(GetPart("MPC555 SBC"), "SPI_OUT", this, "SerialChan_Data_1_Out");
    MakeConnection(GetPart("MPC555 SBC"), "_CS0", this, "_CS_DATA_1");

    // Serial Input Reg signals
    MakeConnection(GetPart("Logic Supply"), "LOGICAL_POWER", GetPart("Serial Input"), "LOGICAL_POWER");
    MakeConnection(this, "SerialChan_1_In",  GetPart("Serial Input"), "SerialChan_In");
    MakeConnection(this, "_CS_1", GetPart("Serial Input"), "_CS");

    // Pseudo AFDX signals
    MakeConnection(GetPart("Logic Supply"), "LOGICAL_POWER", GetPart("Pseudo AFDX"), "LOGICAL_POWER");
    MakeConnection(this, "Databus_A_I", GetPart("Pseudo AFDX"), "Databus_I");
    MakeConnection(GetPart("Pseudo AFDX"), "Databus_O", this, "Databus_A_O");

    // Video ASIC 1 signals
    MakeConnection(GetPart("Logic Supply"), "LOGICAL_POWER", GetPart("Video 1"), "LOGICAL_POWER");
    MakeConnection(GetPart("Video 1"), "RGB_OUT", this, "RGB_1");

    // Video ASIC 2 signals
    MakeConnection(GetPart("Logic Supply"), "LOGICAL_POWER", GetPart("Video 2"), "LOGICAL_POWER");
    MakeConnection(GetPart("Video 2"), "RGB_OUT", this, "RGB_2");
}
```